Verification of EB^3 Specifications using CADP

Dimitris Vekris[1] ⋆ , Frédéric Lang[2], Catalin Dima[1], Radu Mateescu[2]

[1] LACL, Université Paris-Est
61, av. du Général de Gaulle, F-94010 Créteil, France
{Dimitrios.Vekris,Catalin.Dima}@u-pec.fr
[2] Inria Grenoble Rhône-Alpes and LIG – CONVECS team
655, av. de l'Europe, Montbonnot, F-38334 Saint Ismier, France
{Frederic.Lang,Radu.Mateescu}@inria.fr

Abstract. EB^3 is a specification language for information systems. The core of the EB^3 language consists of process algebraic specifications describing the behaviour of the entities in a system, and attribute function definitions describing the entity attributes. The verification of EB^3 specifications against temporal properties is of great interest to users of EB^3. In this paper, we propose a translation from EB^3 to LOTOS NT (LNT for short), a value-passing concurrent language with classical process algebra features. Our translation ensures the one-to-one correspondence between states and transitions of the labelled transition systems corresponding to the EB^3 and LNT specifications. We automated this translation with the EB^{3}2LNT tool, thus equipping the EB^3 method with the functional verification features available in the CADP toolbox.

1 Introduction

The EB^3 method [10] is an event-based paradigm tailored for information systems (ISs). EB^3 has been used in the research projects SELKIS [19] and EB^{3}SEC [17], whose primary aim is the formal specification of ISs with security policies. In the EB^{3}SEC project, real banking industry case studies have been studied, describing interaction with brokers, customers and external financial systems. The SELKIS project deals with two case studies from the medical domain. The first one draws data records from medical imaging devices. The access to these records is done via web-based applications. The second one deals with availability and confidentiality issues for medical emergency units evolving in a great mountain range, like the Alps in that case.

A typical EB^3 specification defines entities, associations, and their respective attributes. The process algebraic nature of EB^3 enables the explicit definition of intra-entity constraints, making them easy for the IS designer to review and understand. Yet, its particular feature compared to classical process algebras,

⋆ Partially supported by SELKIS ANR Project. The LACL group is grateful to CONVECS for its warm welcome to INRIA Grenoble in 2012 and wishes to thank all its members for useful advice and discussions.

such as CSP [15], lies in the use of *attribute functions*, a special kind of recursive functions evaluated on the system execution trace. Combined with guards, attribute functions facilitate the definition of complex inter-entity constraints involving the history of events. The use of attribute functions simplifies system understanding, enhances code modularity, and streamlines maintenance. However, given that ISs are complex systems involving data management and concurrency, a rigorous design process based on formal specification using EB^3 must be completed with effective formal verification features.

Existing attempts for verifying EB^3 specifications are based on translations from EB^3 to other formal methods equipped with verification capabilities. A first line of work [13, 14] focused on devising translations from EB^3 attribute functions and processes to the B method [2], which opened the way for proving invariant properties of EB^3 specifications using tools like Atelier B [6]. Another line of work concerned the verification of temporal logic properties of EB^3 specifications by means of model checking techniques. For this purpose, the formal description and verification of an IS case-study using six model checkers was undertaken in [9, 5]. This study revealed the necessity of branching-time logics for accurately characterizing properties of ISs, and also the fact that process algebraic languages are suitable for describing the behaviour and synchronization of IS entities. However, no attempt of providing a systematic translation from EB^3 to a target language accepted as input by a model checker was made so far.

In this paper, we aim at filling this gap by proposing a translation from EB^3 to LNT [7], a new generation process algebraic specification language inspired from E-LOTOS [16]. As far as we know, this is the first attempt to provide a general translation from EB^3 to a classical value-passing process algebra. It is worth noticing that CSP and LNT were already considered in [9] for describing ISs, and identified as candidate target languages for translating EB^3. Since our primary objective was to provide temporal property verification features for EB^3, we focused our attention on LNT, which is one of the input languages accepted by the CADP verification toolbox [11], and hence is equipped with on-the-fly model checking for action-based, branching-time logics involving data.

At first sight, given that EB^3 has structured operational semantics based on a labelled transition system (LTS) model, its translation to a process algebra may seem straightforward. However, this exercise proved to be rather complex, the main difficulty being to translate a history-based language to a process algebra with standard LTS semantics. To overcome this difficulty, we considered alternative memory-based semantics of EB^3 [20], which were shown to be equivalent to the original trace-based semantics defined for finite-state systems in [10]. Another important ingredient of the translation was the multiway value-passing rendezvous of LNT, which enabled to obtain a one-to-one correspondence between the transitions of the two LTSs underlying the EB^3 and LNT descriptions, and hence to preserve strong bisimulation. The presence of array types and of usual programming language constructs (e.g., loops and conditionals) in LNT was also helpful for specifying the memory, the Kleene star-closure operators, and the EB^3 guarded expressions containing attribute function calls. At last,

$$
\begin{array}{l}
EB^3 ::= A_1; \ldots; A_n; S_1; \ldots; S_m \\
\quad A ::= f\,(\mathsf{T} : \mathcal{T}, \overline{y} : \overline{T}) : T = \textbf{match}\ \mathit{last}\,(\mathsf{T})\ \textbf{with} \\
\qquad\qquad \bot : v_0 \ \mid\ \alpha_1\,(\overline{x_1}) : v_1 \ \mid\ \ldots\ \mid\ \alpha_q\,(\overline{x_q}) : v_q\ [\mid\ \_ : v_{q+1}] \\
\quad S ::= P\,(\overline{x}) = E \\
\quad E ::= \lambda \mid \alpha\,(\overline{v}) \mid E_1.E_2 \mid E_1 \mid E_2 \mid E_0{}^* \mid E_1 \,|[\Delta]|\, E_2 \mid |x\!:\!V\!:\!E_0 \mid \\
\qquad\quad |[\Delta]|\, x\!:\!V\!:\!E_0 \mid C \Rightarrow E \mid P\,(\overline{v})
\end{array}
$$

Fig. 1. EB^3 syntax

the constructed data types and pattern-matching mechanisms of LNT enabled a natural description of EB^3 data types and attribute functions.

We implemented our translation in the EB^{3}2LNT tool, thus making possible the analysis of EB^3 specifications using all the state-of-the-art features of the CADP toolbox, in particular the verification of data-based temporal properties expressed in MCL [18] using the on-the-fly model checker EVALUATOR 4.0.

The paper is organized as follows. Sections 2 and 3 give an overview of the EB^3 and LNT languages, respectively. Section 4 presents our translation from EB^3 to LNT, implemented by the EB^{3}2LNT translator. Section 5 shows how EB^{3}2LNT and CADP can be used for verifying the correctness requirements of an IS. Finally, Section 6 summarizes the results and draws up lines for future work.

2 The Language EB^3

The EB^3 method has been specially designed to specify the functional behaviour of ISs. A standard EB^3 specification comprises (1) a class diagram representing entity types and associations for the IS being specified, (2) a process algebra specification, denoted by *main*, describing the IS, i.e., the valid traces of execution describing its behaviour, (3) a set of attribute function definitions, which are recursive functions on the system execution trace, and (4) input/output rules to specify outputs for input traces, or SQL expressions used to specify queries on the class diagram. We limit the presentation to the process algebra and the set of attribute functions. The EB^3 syntax is presented in Figure 1 and the EB^3 trace semantics Sem_{T} [10] are given in Figure 2 as a set of rules named T_1 to T_{11}. Both figures are commented below.

Process expressions. We write $x, y, x_1, x_2, \ldots$ for variables and $v, w, v_1, v_2, \ldots$ for data expressions over user-defined domains, such as integers, Booleans and more complex domains that we do not give formally, for conciseness. Expressions are built over variables, constants, and standard operations. We also use the overlined notation as a shorthand notation for lists, e.g., $\overline{x}$ denotes a list of variables $x_1, \ldots, x_n$ of arbitrary length. An EB^3 specification consists of a set of attribute function definitions $A_1, \ldots, A_n$, and of a set of process definitions of the form "$P\,(\overline{x}) = E$", where P is a process name and E is a *process expression*.

Let *Act* be a set of *actions* written $\rho, \rho_1, \rho_2, \ldots$ and *Lab* be a set of *labels* written $\alpha, \alpha_1, \alpha_2, \ldots$ Each action ρ is either the *internal action* written λ, or a

$$(\mathrm{T}_1)\ \frac{}{\rho \xrightarrow{\rho} \surd} \qquad (\mathrm{T}_7)\ \frac{E_1 \xrightarrow{\rho} E_1' \quad E_2 \xrightarrow{\rho} E_2'}{E_1\ |[\Delta]|\ E_2 \xrightarrow{\rho} E_1'\ |[\Delta]|\ E_2'}\ in\,(\rho, \Delta)$$

$$(\mathrm{T}_2)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1.E_2 \xrightarrow{\rho} E_1'.E_2} \qquad (\mathrm{T}_8)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1\ |[\Delta]|\ E_2 \xrightarrow{\rho} E_1'\ |[\Delta]|\ E_2}\ \neg in\,(\rho, \Delta)$$

$$(\mathrm{T}_3)\ \frac{E_2 \xrightarrow{\rho} E_2'}{\surd.E_2 \xrightarrow{\rho} E_2'} \qquad (\mathrm{T}_9)\ \frac{}{\surd\ |[\Delta]|\ \surd \xrightarrow{\lambda} \surd}$$

$$(\mathrm{T}_4)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1 \mid E_2 \xrightarrow{\rho} E_1'} \qquad (\mathrm{T}_{10})\ \frac{E_0 \xrightarrow{\rho} E_0'}{C \Rightarrow E_0 \xrightarrow{\rho} E_0'}\ \|C\|$$

$$(\mathrm{T}_5)\ \frac{}{{E_0}^* \xrightarrow{\lambda} \surd} \qquad (\mathrm{T}_{11})\ \frac{E[\overline{x} := \overline{v}] \xrightarrow{\rho} E'}{P\,(\overline{v}) \xrightarrow{\rho} E'}\ P\,(\overline{x}) = E$$

$$(\mathrm{T}_6)\ \frac{E_0 \xrightarrow{\rho} E_0'}{{E_0}^* \xrightarrow{\rho} E_0'.{E_0}^*}$$

Fig. 2. EB^3 trace semantics Sem_{T}

visible action of the form "$\alpha\,(\overline{v})$", where $\alpha \in Lab$. An action ρ is the simplest process expression, whose semantics are given by rule T_1. The symbol $\surd$ (which is not part of the user syntax) denotes successful execution. The *trace* T (implicit in the presentation) of an EB^3 specification at a given moment consists of the sequence of visible actions executed since the start of the system. (Note therefore that λ does not appear in the trace.) At system start, the trace is empty. If T denotes the current trace and action ρ can be executed, then $\mathsf{T}.\rho$ denotes the trace just after executing ρ.

EB^3 processes can be combined with classical process algebra operators such as the *sequence* "$E_1.E_2$" ($\mathrm{T}_2, \mathrm{T}_3$), the *choice* "$E_1 \mid E_2$" ($\mathrm{T}_4$) and the *Kleene* closure "${E_0}^*$" ($\mathrm{T}_5, \mathrm{T}_6$). Rules ($\mathrm{T}_7$ to T_9) define *parallel* composition "$E_1\ |[\Delta]|\ E_2$" of E_1, E_2 with synchronization on $\Delta \subseteq Lab$. The condition "$in\,(\rho, \Delta)$" is true iff the label of ρ belongs to Δ. The symmetric rules for choice and parallel composition have been omitted for brevity. Expressions "$E_1\ |||\ E_2$" and "$E_1\ ||\ E_2$" are equivalent respectively to "$E_1\ |[\emptyset]|\ E_2$" and "$E_1\ |[Lab]|\ E_2$".

The *guarded expression* process "$C \Rightarrow E_0$" (T_{10}) can execute E_0 if the Boolean condition C holds, which is denoted by the side condition "$\|C\|$". Since C may contain calls to attribute functions, its evaluation depends on the trace obtained up to the moment when the condition is evaluated. Note that the evaluation of the guard C and the execution of the first action ρ in E_0 are simultaneous, i.e., no action is allowed in concurrent processes in the meantime. We call this property the *guard-action atomicity*. This property is essential for consistency as, by side effects, the occurrence of actions in concurrent processes could implicitly change the value of C before the guarded action has been executed.

Quantification is permitted for *choice* and *parallel* composition. If V is a set of expressions $\{v_1, \ldots, v_n\}$, "$\mid x : V : E_0$" and "$\mid[\Delta]\mid x : V : E_0$" stand respectively for "$E_0[x := v_1] \mid \ldots \mid E_0[x := v_n]$" and "$E_0[x := v_1] \mid[\Delta]\mid \ldots \mid[\Delta]\mid E_0[x := v_n]$", where "$E[x := v]$" denotes the replacement of all occurrences of x by v in E. For instance, "$\mid\mid x : \{1, 2, 3\} : a\,(x)$" stands for "$a\,(1) \mid\mid a\,(2) \mid\mid a\,(3)$". At last, named processes can be instantiated as usual (T_{11}). Given an EB^3 process expression E, we write $vars\,(E)$ for the set of variables occurring free in E.

Attribute functions. Attribute function definitions are denoted by the symbol A in the grammar of Figure 1. Attribute functions are defined recursively on the current trace T representing the history of actions executed, with the aid of functions $last\,(\mathsf{T})$ which denotes the last action of the trace, and $front\,(\mathsf{T})$ which denotes the trace without its last action. The symbol $\perp$ represents the undefined value. In particular, both $last\,(\mathsf{T})$ and $front\,(\mathsf{T})$ match $\perp$ when the trace is empty. The symbol _ (wildcard) matches all actions not matched by any of the preceding action patterns $\alpha_1\,(\overline{x_1}), \ldots, \alpha_q\,(\overline{x_q})$. Each v_i $(i \in 0..n)$ is an expression of the same type as f's return type built over the variables $\overline{y} \cup \overline{x_i}$.

For defining formal semantics for attribute functions, the rule system of Figure 2 has to be expanded with trace and memory contexts for each process, representing the sequence of actions executed since the process was initiated, and the value of attribute functions for the current trace and any value for the rest of their arguments, stored into process memory M. Due to space limitations, we do not present the formal semantics here, but show how attribute functions are evaluated on a concrete example. The formal trace-memory semantics for attribute functions can be found in the companion paper [20].

Example. We give an example of how the trace-memory semantics work for a simplified library management system, whose specification (processes and attribute functions) in EB^3 is given in Figure 3. Process *main* is the parallel interleaving between m instances of process *book* and p instances of process *member*. Process *book* stands for a book acquisition followed by its eventual discard. The attribute function "$borrower\,(\mathsf{T},\ bId)$" looks for actions of the form "$Lend\,(mId, bId)$" or "$Return\,(bId)$" in the trace and returns the current borrower of book bId or $\perp$ if the book is not lent. In process book, action "$Discard\,(bId)$" is thus guarded to guarantee that book bId cannot be discarded if it is currently lent. How the use of attribute functions enhances expressiveness in the EB^3 specification of Figure 3 is discussed in [20].

We illustrate how the EB^3 specification describing the library management system is evaluated. The idea lies in the observation that attribute functions can be turned into state variables (the memory M) carrying the effect of the system trace on their corresponding values. This avoids keeping the (ever-growing) trace leading to a finite state model. If $f\,(\mathsf{T}, x_1 : T_1, \ldots, x_l : T_l)$ is an attribute function, we construct $|T_1| \times \ldots \times |T_l|$ state variables, where $|T_i|$ $(i \in 1..l)$ stands for T_i's cardinality.

As an example, we set $m = p = NbLoans = 2$, i.e. we consider two books b_1 and b_2, and two members m_1 and m_2. The memory has four cells: $\mathsf{M} =$

```
BID = {b_1, ..., b_m}, MID = {m_1, ..., m_p}
book (bId : BID) =
    Acquire (bId) . (borrower (T, bId) = ⊥) ⇒ Discard (bId)
loan (mId : MID, bId : BID) =
    (borrower (T, bId) = ⊥) ∧ (nbLoans (T, mId) < NbLoans) ⇒
        Lend (bId, mId) . Return (bId)
member (mId : MID) =
    Register (mId) . (||| bId : BID : loan (mId, bId)*) . Unregister (mId)
main =
    (||| bId : BID : book (bId)*) ||| (||| mId : MID : member (mId)*)
```

```
nbLoans (T : 𝒯, mId : MID) : Nat_⊥ =
   match last (T) with
     ⊥ : ⊥
   | Lend (bId, mId) :
         nbLoans (front (T), mId) + 1
   | Register (mId) : 0
   | Unregister (mId) : ⊥
   | Return (bId) :
       if mId = borrower (T, bId) then
         nbLoans (front (T), mId) − 1
       else nbLoans (front (T), mId) end if
   | _ : nbLoans (front (T), mId)
   end match
```

```
borrower (T : 𝒯, bId : BID) : MID_⊥ =
   match last (T) with
     ⊥ : ⊥
   | Lend (bId, mId) : mId
   | Return (bId) : ⊥
   | _ : borrower (front (T), bId)
   end match
```

Fig. 3. EB^3 specification of a library management system

(*borrower* [b_1], *borrower* [b_2], *nbLoans* [m_1], *nbLoans* [m_2]). The first two cells keep the two values of the attribute function *borrower* (T, •) for a given trace T, and the last two keep the values of *nbLoans* (T, •). After every step, the new value of each cell can be calculated from the previous memory and the action that has just been executed. The memory is initially set to $(\bot, \bot, \bot, \bot)$. After the trace "*Acquire* (b_1).*Acquire* (b_2).*Register* (m_1).*Register* (m_2)" the memory contains $(\bot, \bot, 0, 0)$. If action "*Lend* (b_1, m_1)" is then executed, the new memory is $(m_1, \bot, 1, 0)$. For instance, the new value m_1 for *borrower* [b_1] is obtained from the rule "*Lend* (*bId*, *mId*) : *mId*" in the definition of the attribute function *borrower* (see Fig. 3), and the new value 1 for *nbLoans* [m_1] by the rule "*Lend* (*bId*, *mId*) : *nbLoans* (*front* (T), *mId*) + 1" of the attribute function *nbLoans*, where the value of *nbLoans* (*front* (T), m_1) corresponds to the value of *nbLoans* [m_1] in the previous memory state (value 0).

3 The Language LNT

LNT aims at providing the best features of imperative and functional programming languages and value-passing process algebras. It has a user friendly syntax and formal operational semantics defined in terms of labeled transition systems

$$
\begin{array}{|l|}
\hline
B ::= \textbf{stop} \mid \textbf{null} \mid G\,(O_1,\ldots,O_n)\ \textbf{where}\ E \mid B_1; B_2 \\
\quad\ \mid\ \textbf{if}\ E\ \textbf{then}\ B_1\ \textbf{else}\ B_2\ \textbf{end if} \mid \textbf{var}\ x\!:\!T\ \textbf{in}\ B\ \textbf{end var} \mid x := E \mid \\
\quad\ \mid\ \textbf{loop}\ L\ \textbf{in}\ B\ \textbf{end loop} \mid \textbf{break}\ L \mid \textbf{select}\ B_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ B_n\ \textbf{end select} \\
\quad\ \mid\ \textbf{par}\ G_1,\ldots,G_n\ \textbf{in}\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{end par} \mid P\,\texttt{[}G_1,\ldots,G_n\texttt{]}\,(E_1,\ldots,E_n) \\
\hline
O ::= \texttt{!}E \mid\ \texttt{?}x \\
\hline
\end{array}
$$

Fig. 4. LNT syntax (limited to the fragment used in this paper)

(LTSs). LNT is supported by the LNT.OPEN tool of CADP, which allows the on-the-fly exploration of the LTS corresponding to an LNT specification.

We present the fragment of LNT that serves as the target of our translation. Its syntax is given in Figure 4. LNT terms denoted by B are built from actions, choice (**select**), conditional (**if**), sequential composition (**;**), breakable loop (**loop** and **break**) and parallel composition (**par**). Communication is carried out by rendezvous on gates, written G, G_1, ..., G_n, and may be guarded using Boolean conditions on the received values (**where** clause). LNT allows multiway rendezvous with bidirectional (send/receive) value exchange on the same gate occurrence, each offer O being either a send offer (!) or a receive offer (?), independently of the other offers. Expressions E are built from variables, type constructors, function applications and constants. Labels L identify loops, which can be escaped using "**break** L" from inside the loop body. Processes are parameterized by gates and data variables. LNT semantics are formally defined in SOS style in [7].

4 Translation from EB^3 to LNT

Principles. Our translation of EB^3 relies on the trace-memory semantics. Thus, we explicitly model in LNT a memory, which stores the state variables corresponding to attribute functions (we call these variables *attribute variables*) and is modified each time an action is executed.

Assuming n attribute functions $f_1,\ldots,f_n$, we model the memory as a process M placed in parallel with the rest of the system (a common approach for modeling global variables in process algebras), which manages for each attribute function f_i an attribute variable (also named f_i) that encodes the function. To read the values of these attribute variables (i.e., to evaluate the attribute functions), processes need to communicate with the memory M, and every action must have an immediate effect on the memory (so as to reflect the immediate effect on the execution trace). To achieve this, the memory process synchronizes with the rest of the system on every possible action of the system (including λ, to which we associate an LNT gate also written λ in abstract syntax for convenience), and updates its attribute variables accordingly. The list of attribute variables $\overline{f} = (f_1,\ldots,f_n)$ is added as a supplementary offer on each EB^3 action $\alpha\,(\overline{v})$, so that attribute variables can be directly accessed to evaluate the guard associated to the action, wherever needed, while guaranteeing the guard-action atomicity. Therefore, every action $\alpha\,(\overline{v})$ will be encoded in LNT as $\alpha\,(!\overline{v}, ?\overline{f})$, and synchronized with an action of the form $\alpha\,(?\overline{x}, !\overline{f})$ in the memory process M, thus taking benefit of bidirectional value exchange during the rendezvous.

Translation of attribute functions. To formalize the translation, we assume $Lab = \{\alpha_1, \ldots, \alpha_q\}$ (not including λ), each α_j has formal parameters $\overline{x_j}$, $\{f_1, \ldots, f_n\}$ is the set of attribute functions, and each f_i is uniquely defined by the set of formal parameters $\overline{y_i}$ and the set of data expressions $w_i^0, \ldots, w_i^q$, such that:

$$f_i\,(\mathsf{T}, \overline{y_i}) = \textbf{match}\ \mathit{last}\,(\mathsf{T})\ \textbf{with}\ \bot : w_i^0 \mid \alpha_1\,(\overline{x_1}) : w_i^1 \mid \ldots \mid \alpha_q\,(\overline{x_q}) : w_i^q$$

We also assume that the attribute functions are ordered, so that for all $h \in 1..n, i \in 1..n, j \in 1..q$, every function call of the form $f_h\,(\mathsf{T}, \ldots)$ occurring in w_i^j satisfies $h < i$ and every call of the form $f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$ satisfies $h \geq i$. Such an ordering can be constructed if the EB^3 specification does not contain circular dependencies between function calls, which would potentially lead to infinite attribute function evaluation. In particular, the definition of an attribute function f_i cannot contain recursive calls of the form "$f_i\,(\mathsf{T}, \ldots)$", but only recursive calls of the form "$f_i\,(\mathit{front}\,(\mathsf{T}), \ldots)$". Note that this does not limit the expressiveness of EB^3 attribute functions, because every recursive computation on data expressions only (which keeps the trace unchanged) can be described using standard functions and not attribute functions.

Ordering attribute functions in this way allows the memory to be updated consistently, from f_1 to f_n in turn. At every instant, already-updated values correspond to calls of the form $f_h\,(\mathsf{T}, \ldots)$ (the value of f_h on the current trace), whereas calls of the form $f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$ are replaced by accesses to a copy $\overline{f'}$ of the memory $\overline{f}$, which was made before starting the update. This encoding thus enables the trace parameter to be discharged from function calls, ensuring that while updating f_i, accesses to f_h with $h < i$ necessarily correspond to calls with parameter T.

Process M is defined in Figure 5. It runs an infinite loop, which "listens" to all possible actions α_j of the system. Each attribute variable f_i is an array with l_i dimensions, where l_i is the arity of the attribute function f_i minus 1 (because the trace parameter is now discharged). Each dimension of the array f_i thus corresponds to one formal parameter in $\overline{y_i}$, so that $f_i\,[\textbf{ord}\,(v_1)] \ldots [\textbf{ord}\,(v_{l_i})]$ encodes the current value of $f_i\,(\mathsf{T}, v_1, \ldots, v_{l_i})$, where $\textbf{ord}\,(v)$ is a predefined LNT function that denotes the *ordinate* of value v, i.e., a unique number between 1 and the cardinal of v's type. For each type T we assume the existence of functions first_T that returns the first element of type T, last_T that returns the last element of type T, and $\mathit{next}_T\,(x)$ that returns the successor of x in type T (following the total order induced by **ord**). Such functions are available in LNT for all finite types. Function *mod* transforms an expression E by syntactically replacing function calls by array accesses, while discharging the trace parameter as explained above.

Upon synchronisation on action $\alpha_j\,(?\overline{x_j}, !\overline{f})$ with the LNT process corresponding to EB^3's *main* process (see translation of processes below), the values of all attribute variables f_i ($i \in 1..n$) are updated using function upd_i^j.

```
process M [α_1, ..., α_q, λ : any] is
  var f̄, f̄' : type (f̄),
      ȳ_1 : type (ȳ_1), ..., ȳ_n : type (ȳ_n), x̄_1 : type (x̄_1), ..., x̄_q : type (x̄_q) in
    upd_1^0; ... ; upd_n^0;
    loop
      f̄' := f̄ (* f_i' [ord (v̄)] will encode f_i (front (T), v̄) during memory update *)
      select
      α_1 (?x̄_1, !f̄); upd_1^1; ... ; upd_n^1
      [] ... []
      α_q (?x̄_q, !f̄); upd_1^q; ... ; upd_n^q
      [] λ (!f̄)
      end select
    end loop
  end var
end process
upd_i^j ≐ enum (ȳ_i, f_i [ord (ȳ_i)] := mod (w_i^j))
enum ([ ], B) ≐ B
enum (x :: ȳ, B) ≐ x := first_T;
                  loop L_x in
                    enum (ȳ, B)
                    if x ≠ last_T then x := next_T (x) else break L_x end if
                  end loop where T = type (x)
v [ord (ȳ)] ≐ v [ord (y_1)] ... [ord (y_l)], ?ȳ = (?y_1, ..., ?y_l), where ȳ = (y_1, ..., y_l)
mod (E) ≐ E [ f_i (T, v̄_i) := f_i [ord (v̄_i)], f_i (front (T), v̄_i) := f_i' [ord (v̄_i)] | i ∈ 1..n ]
```

Fig. 5. LNT code for the memory process implementing attribute functions

Translation of processes. We define a translation function t from an EB^3 process expression to an LNT process. Most EB^3 constructs are process algebra constructs with a direct correspondence in LNT. The main difficulty arises in the translation of guarded process expressions of the form "$C \Rightarrow E_0$" in a way that guarantees the guard-action atomicity. This led us to consider a second parameter for the translation function t, namely the condition C, whose evaluation is delayed until the first action occurring in the process expression E_0. The definition of $t\,(E, C)$ is given in Figure 6. An EB^3 specification E_0 will then be translated into "**par** $\alpha_1, \ldots, \alpha_q, \lambda$ **in** $t\,(E_0, \text{true})$ || M [$\alpha_1, \ldots, \alpha_q, \lambda$] **end par**" and every process definition of the form "$P\,(\overline{x}) = E$" will be translated into the process "**process** P [$\alpha_1, \ldots, \alpha_q, \lambda$: **any**] ($\overline{x}$: type ($\overline{x}$)) **is** $t\,(E, \text{true})$ **end process**", where $\{\alpha_1, \ldots, \alpha_q\} = Lab$. The rules of Figure 6 can be commented as follows:

- Rule (1) translates the λ action. Note that λ cannot be translated to the empty LNT statement **null**, because execution of λ may depend on a guard C, whose evaluation requires the memory to be read, so as to get attribute variable values. This is done by the LNT communication action $\lambda\,(?\overline{f})$. The guard C is evaluated after replacing calls to attribute functions (all of which have the form $f_i\,(\mathsf{T}, \overline{v_i})$) by the appropriate attribute variables, using function *mod* defined in Figure 5. Rule (2) is similar but handles visible actions.

$$
\begin{aligned}
t\,(\lambda, C) &= \lambda\,(?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) && (1)\\
t\,(\alpha\,(\overline{v}), C) &= \alpha\,(\overline{v}, ?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) && (2)\\
t\,(E_1.E_2, C) &= t\,(E_1, C);\ t\,(E_2, \mathrm{true}) && (3)\\
t\,(C' \Rightarrow E_0, C) &= t\,(E_0, C\ \textbf{andthen}\ C') && (4)\\
t\,(E_1 \mid E_2, C) &= \textbf{select}\ t\,(E_1, C)\ \texttt{[]}\ t\,(E_2, C)\ \textbf{end select} && (5)\\
t\,(\mid x : V : E_0, C) &= \textbf{var}\ x := \textbf{any}\ V;\ t\,(E_0, C)\ \textbf{end var} && (6)\\
t\,({E_0}^*, \mathrm{true}) &= \textbf{loop}\ L_{E_0}\ \textbf{in}\\
&\qquad \textbf{select}\\
&\qquad\quad \lambda\,(?\overline{f});\ \textbf{break}\ L_{E_0}\ \texttt{[]}\ t\,(E_0, \mathrm{true})\\
&\qquad \textbf{end select}\\
&\quad \textbf{end loop} && (7)\\
t\,(E_1\ |[\Delta]|\ E_2, \mathrm{true}) &= \textbf{par}\ \Delta\ \textbf{in}\ t\,(E_1, \mathrm{true})\ ||\ t\,(E_2, \mathrm{true})\ \textbf{end par} && (8)\\
t\,(|[\Delta]|\,x : V : E_0, \mathrm{true}) &= \textbf{par}\ \Delta\ \textbf{in}\ E_0[x := v_1]\ ||\ \ldots\ ||\ E_0[x := v_n]\ \textbf{end par}\\
&\qquad \text{where } V = \{v_1, \ldots, v_n\} && (9)\\
t\,(P\,(\overline{v}), \mathrm{true}) &= P\ [\alpha_1, \ldots, \alpha_q, \lambda]\ (\overline{v}) && (10)
\end{aligned}
$$

In all other cases:

$$
t\,(E_0, C) = \begin{cases}
\textbf{if}\ \mathit{mod}\,(C)\ \textbf{then}\ t\,(E_0, \mathrm{true})\ \textbf{else stop end if} \\
\quad \text{if } C \text{ does not use attribute functions} \\
\textbf{par}\ \alpha_1, \ldots, \alpha_q, \lambda\ \textbf{in} \\
\quad t\,(E_0, \mathrm{true}) \\
||\ \mathit{pr}_C\ [\alpha_1, \ldots, \alpha_q, \lambda]\ (\mathit{vars}\,(C)) \\
\textbf{end par} \quad \text{otherwise}
\end{cases} \qquad (11)
$$

Fig. 6. Translation from EB^3 process to LNT process

- Rule (3) translates EB^3 sequential composition into LNT sequential composition, passing the evaluation of C to the first process expression.
- Rule (4) makes a conjunction between the guard of the current process expression with the guard already accumulated from the context.
- Rules (5) and (6) translate the choice and quantified choice operators of EB^3 into their direct LNT counterpart.
- Rule (7) translates the Kleene closure into a combination of LNT loop and select, following the identity ${E_0}^* = \lambda \mid E_0.{E_0}^*$.
- Rule (8) translates EB^3 parallel composition into LNT parallel composition.
- Rule (9) translates EB^3 quantified parallel composition into LNT parallel composition by expanding the type V of the quantification variable, since LNT does not have a quantified parallel composition operator.
- Rule (10) translates an EB^3 process call into the corresponding LNT process call, which requires gates to be passed as parameters.
- Rules (7) to (10) only apply when the guard C is trivially true. In the other cases, we must apply rule (11), which generates code implementing the guard. If C does not use attribute functions, i.e., does not depend on the trace, then it can be evaluated immediately without communicating with the memory

process (first case). Otherwise, the guard evaluation must be delayed until the first action of the process expression E_0. When E_0 is either a Kleene closure, a parallel composition, or a process call, identifying its first action syntactically is not obvious. One solution would consist in expanding E_0 into a choice in which every branch has a fixed initial action[3], to which the guard would be added. We preferred an alternative solution that avoids the potential combinatorial explosion of code due to static expansion. A process pr_C (defined in Fig. 7) is placed in parallel to $t\,(E_0, \text{true})$ and both processes synchronize on all actions. Process pr_C imposes on $t\,(E_0, \text{true})$ the constraint that the first executed action must satisfy the condition C (**then** branch). For subsequent actions, the condition is relaxed (**else** branch).

The following example illustrates and justifies the use of process pr_C as a means to solve the guard-action atomicity problem. Consider the EB[3] system "$C \Rightarrow \mathit{Lend}\,(b_1,\ m_1)\ |||\ \mathit{Return}\,(b_2)$", where C denotes the Boolean condition "$\mathit{borrower}\,(\mathsf{T},\ b_1) = \bot \wedge \mathit{nbLoans}\,(\mathsf{T},\ m_1) < \mathit{NbLoans}$" and $\mathit{Lab} = \{\mathit{Lend}, \mathit{Return}\}$. The LNT code corresponding to this system is the following:

$$
\begin{array}{l}
\textbf{par}\ \mathit{Lend}, \mathit{Return}, \lambda\ \textbf{in} \\
\quad \textbf{par}\ \mathit{Lend}, \mathit{Return}, \lambda\ \textbf{in} \\
\quad\quad \textbf{par}\ \mathit{Lend}\,(b_1,\ m_1, ?\overline{f})\ ||\ \mathit{Return}\,(b_2, ?\overline{f})\ \textbf{end par} \\
\quad ||\ pr_C\ [\mathit{Lend}, \mathit{Return}, \lambda]\ (b_1,\ m_1) \\
\quad \textbf{end par} \\
||\ M\ [\mathit{Lend}, \mathit{Return}, \lambda] \\
\textbf{end par}
\end{array}
$$

The first action executed by this system may be either *Lend* or *Return*. We consider the case where *Lend* is executed first. According to the LNT semantics, it results from the multiway synchronization of the following three actions:

- "$\mathit{Lend}\,(b_1,\ m_1, ?\overline{f})$" in the above expression,
- "$\mathit{Lend}\,(?b, ?m, ?\overline{f})$ **where** $\mathit{borrower}[\mathbf{ord}(b_1)] = \bot \wedge \mathit{nbLoans}[\mathbf{ord}(m_1)] < \mathit{NbLoans}$" in process pr_C (at this moment, *start* is true, see Fig. 7), and
- "$\mathit{Lend}\,(?b, ?m, !\overline{f})$" in process M (see Fig. 5).

Thus, in pr_C at synchronization time, $\overline{f}$ is an up-to-date copy of the memory stored in M, $b = b_1$, and $m = m_1$. The only condition for the synchronization to occur is the guard $\mathit{mod}\,(C)$, whose value is computed using the up-to-date copy $\overline{f}$ of the memory. In case $\mathit{mod}\,(C)$ evaluates to true, no other action (susceptible to modifying $\overline{f}$) can occur between the evaluation of $\mathit{mod}\,(C)$ and the occurrence of *Lend* as both happen synchronously, thus achieving the guard-action atomicity. Once *Lend* has occurred, *Return* can occur without any condition, as the value of *start* has now become false.

[3] Such a form, commonly called *head normal form* [3], is used principally in the context of the process algebra ACP [4] to analyse the behaviour of recursive processes.

```
process pr_C [α_1, ..., α_q, λ : any] (vars (C) : type (vars (C))) is
var start : bool, x̄_1 : type (x̄_1), ..., x̄_q : type (x̄_q), f̄ : type (f̄) in
  start := true;
  loop L in select
    if start then
      start := false;
      select
        α_1 (?x̄_1, ?f̄) where mod (C)
        [] ... []
        α_q (?x̄_q, ?f̄) where mod (C)
        []
        λ (?f̄) where mod (C)
      end select
    else
      select
        α_1 (?x̄_1, ?f̄)
        [] ... []
        α_q (?x̄_q, ?f̄)
        []
        λ (?f̄)
      end select
    end if
  [] break L end select end loop
end var
end process
```

Fig. 7. Process pr_C

Theorem 1. *Let E, E' be* EB^3 *process expressions, T be the current trace, $\overline{f}$ be the set of attribute functions, and $\rho \in Act$. Then $E \xrightarrow{\rho\,(\overline{x})} E'$ if and only if:*

$$t\,(E, true) \xrightarrow{\rho\,(\overline{x}, \overline{f})} t\,(E', true) \wedge (\forall f_i \in \overline{f})\ (\forall \overline{v})\ f_i\,(\mathsf{T}, \overline{v}) = f_i\,[ord(\overline{v})]\,.$$

The proof strategy for Theorem 1 relies on the existence of a bisimulation between each EB^3 specification and its corresponding LNT translation. It works by providing a match between the reduction rules of EB^3 [20] and the corresponding LNT rules [7].

We developed an automatic translator tool from EB^3 specifications to LNT, named EB^{3}2LNT, implemented using the Ocaml Lex/Yacc compiler construction technology. It consists of about 900 lines of OCaml code. We applied EB^{3}2LNT on a benchmark of EB^3 specifications, which includes variations of the library management system (examined in its simplest version in Section 2) and a bank account management system.

We noticed that, for each EB^3 specification, the code size of the equivalent LNT specification is twice as big. Part of this expansion is caused by the fact that LNT is more structured than EB^3: LNT requires more keywords and gates have to be declared and passed as parameters to each process call. By looking at the rules of Figure 6, we can see that the other causes of expansion are rule (5), which duplicates the condition C, and rule (9), which duplicates the body E_0 of the quantified parallel composition operator "$|[\Delta]|\, x : V : E_0$" as many times as there are elements in the set V. Both expansions are linear in the size of the source EB^3 code. However, in the case of a nested parallel composition "$|[\Delta_1]|\, x_1 : V_1 : \ldots |[\Delta_n]|\, x_n : V_n : E_0$", the expansion factor is as high as the product of the number of elements in the respective sets $V_1, \ldots, V_n$, which may be large. If E_0 is a big process expression, the expansion can be limited by encapsulating E_0 in a parameterized process "$P_{E_0}\,(x_1, \ldots, x_n)$" and replacing duplicated occurrences of E_0 by appropriate instances of P_{E_0}.

5 Case Study

We illustrate below the application of the EB^{3}2LNT translator in conjunction with CADP for analyzing an extended version of the IS library management system, whose description in EB^3 can be found in Annex C of [12]. With respect to the simplified version presented in Section 2, the IS enables e.g., members to renew their loans and to reserve books, and their reservations to be cancelled or transferred to other members on demand. The desired behaviour of this IS was characterized in [9] as a set of 15 requirements expressed informally as follows:

R1. A book can always be acquired by the library when it is not currently acquired.
R2. A book cannot be acquired by the library if it is already acquired.
R3. An acquired book can be discarded only if it is neither borrowed nor reserved.
R4. A person must be a member of the library in order to borrow a book.
R5. A book can be reserved only if it has been borrowed or already reserved by some member.
R6. A book cannot be reserved by the member who is borrowing it.
R7. A book cannot be reserved by a member who is reserving it.
R8. A book cannot be lent to a member if it is reserved.
R9. A member cannot renew a loan or give the book to another member if the book is reserved.
R10. A member is allowed to take a reserved book only if he owns the oldest reservation.
R11. A book can be taken only if it is not borrowed.
R12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
R13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or cancelled.
R14. Ultimately, there is always a procedure that enables a member to leave the library.
R15. A member cannot borrow more than the loan limit defined at the system level for all users.

We expressed all the above requirements using the property specification language MCL [18]. MCL is an extension of the alternation-free modal μ-calculus [8] with action predicates enabling value extraction, modalities containing extended regular expressions on transition sequences, quantified variables and parameterized fixed point operators, programming language constructs, and fairness operators encoding generalized Büchi automata. These features make possible a concise and intuitive description of safety, liveness, and fairness properties involving data, without sacrificing the efficiency of on-the-fly model checking, which has a linear-time complexity for the dataless MCL formulas [18].

We show below the MCL formulation of two requirements from the list above, which denote typical safety and liveness properties. Requirement R2 is expressed in MCL as follows:

$$[\,\mathbf{true}^*.\{\texttt{ACQUIRE ?}B:\mathbf{string}\}.(\mathbf{not}\ \{\texttt{DISCARD !}B\})^*.\{\texttt{ACQUIRE !}B\}\,]\ \mathbf{false}$$

This formula uses the standard *safety* pattern "$[\beta]$ **false**", which forbids the existence of transition sequences matching the regular formula β. Here the undesirable sequences are those containing two *Acquire* operations for the same book B without a *Discard* operation for B in the meantime. The regular formula $\mathbf{true}^*$ matches a subsequence of (zero or more) transitions labeled by arbitrary actions. Note the use of the construct "$\texttt{?}B:\mathbf{string}$", which matches any string and extracts its value in the variable B used later in the formula. Therefore, the above formula captures all occurrences of books carried by *Acquire* operations in the model. Requirement R12 is formulated in MCL as follows:

$$\begin{array}{l}[\,\mathbf{true}^*.\{\texttt{RESERVE ?}M:\mathbf{string}\ \texttt{?}B:\mathbf{string}\}.\\ \ (\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*\,]\\ \quad\langle\,(\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*.\ \{\texttt{CANCEL !}M\ \texttt{!}B\}\,\rangle\ \mathbf{true}\end{array}$$

This formula denotes a *liveness* property of the form "$[\beta_1]\ \langle\beta_2\rangle$ **true**", which states that every transition sequence matching the regular formula β_1 (in this case, book B has been reserved by member M and subsequently neither taken nor transferred) ends in a state from which there exists a transition sequence matching the regular formula β_2 (in this case, the reservation can be cancelled before being taken or transferred).

Using EB^{3}2LNT, we translated the EB^3 specification of the library management system to LNT. The resulting specification was checked against all the 15 requirements, formulated in MCL, using the EVALUATOR 4.0 model checker of CADP. The experiments were performed on an Intel(R) Core(TM) i7 CPU 880 at 3.07GHz. Table 1 shows the results for several configurations of the IS, obtained by instantiating the number of books (m) and members (p) in the IS. All requirements were shown to be valid on the IS specification. The second and third line of the table indicate the number of states and transitions of the LTS corresponding to the LNT specification. The fourth line gives the time needed to generate the LTS and the other lines give the verification time for each requirement. Note that the number of states generated increases with the size of m and p as EVALUATOR 4.0 applies explicit techniques for state space generation.

6 Conclusion

We proposed an approach for equipping the EB^3 method with formal verification capabilities by reusing already available model checking technology. Our

Table 1. Model checking results for the library management system

(m, p)	(3,2)	(3,3)	(3,4)	(4,3)
states	1,002	182,266	8,269,754	27,204,016
trans.	5,732	1,782,348	105,481,364	330,988,232
time	1.9s	14.4s	31'39s	140'22s
R1	0.3s	1.8s	5'19s	20'13s
R2	0.2s	2.9s	9'26s	36'7s
R3	0.2s	9.4s	97'46s	26'47s
R4	0.2s	1.7s	5'15s	18'40s
R5	0.2s	2.2s	6'46s	21'52s
R6	0.2s	4.1s	38'30s	10'19s
R7	0.2s	7.4s	65'22s	24'33s
R8	0.2s	2.2s	6'52s	22'27s
R9	0.2s	2.3s	6'38s	22'29s
R10	0.3s	13.3s	43'59s	62'07s
R11	0.3s	2.5s	6'36s	22'14s
R12	0.3s	4.0s	10'47s	45'09s
R13	0.4s	4.3s	11'46s	1'07s
R14	0.3s	3.6s	10'41s	37'33s
R15	0.2s	2.8s	7'53s	28'56s

approach relies upon a new translation from EB^3 to LNT, which provides a direct connection to all the state-of-the-art verification features of the CADP toolbox. The translation, based on alternative memory semantics of EB^3 [20] instead of the original trace semantics [10], was automated by the EB^{3}2LNT translator and validated on several examples of typical ISs. So far, we experimented only the model checking of MCL data-based temporal properties on EB^3 specifications. However, CADP also provides extensive support for equivalence checking and compositional LTS construction, which can be of interest to IS designers.

As future work, we plan to provide a formal proof of the translation from EB^3 to LNT, which could serve as reference for translating EB^3 to other process algebras as well. We also plan to study abstraction techniques for verifying properties regardless of the number of entity instances that participate in the IS, following the approaches for parameterized model checking [1]. In particular, we will observe how the insertion of new functionalities into an IS affects this issue, and we will formalize this in the context of EB^3 specifications.

References

1. P.A. Abdulla, A. Bouajjani, B. Jonsson, M. Nilsson. Handling Global Conditions in Parameterized System Verification. In *Proc. CAV*, LNCS vol. 1633, pages 134–145, Springer, 1999.
2. J.-R. Abrial. *The B-Book - Assigning programs to meanings.* Cambridge University Press, 2005.
3. J.A. Bergstra, A. Ponse, S.A. Smolka. *Handbook of Process Algebra.* Elsevier, 2001.

4. J.A. Bergstra, J. W. Klop. Algebra of Communicating Processes with Abstraction. *TCS*, 37:77–121, 1985.
5. R. Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke, 2010.
6. ClearSy. *Atelier B.* http://www.atelierb.societe.com.
7. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding. *Reference Manual of the LOTOS NT to LOTOS Translator – Version 5.4.* INRIA/VASY, 2011.
8. E. Allen Emerson, C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proc. of LICS*, pages 267–278, 1986.
9. M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, M. Ouenzar. Comparison of model checking tools for information systems. In *Proc. of ICFEM*, LNCS vol. 6447, pages 581–596, Springer, 2010.
10. M. Frappier, R. St.-Denis. EB^3: an entity-based black-box specification method for information systems. *Software and System Modeling* 2(2):134–149, Springer, 2003.
11. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. of TACAS*, LNCS vol. 6605, pages 372–387, Springer, 2011.
12. F. Gervais. *Combinaison de spécifications formelles pour la modélisation des systèmes d'information*. PhD thesis, Université de Sherbrooke, 2006.
13. F. Gervais, M. Frappier, R. Laleau. Synthesizing B Specifications from EB^3 Attribute Definitions. In *Proc. of iFM*, LNCS vol. 3771, pages 207–226 Springer, 2005.
14. F. Gervais, M. Frappier, R. Laleau. Refinement of EB^3 Process Patterns into B Specifications. In *Proc. of Formal Specification and Development in B*, LNCS vol. 4355, pages 201–215, Springer, 2006.
15. C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.
16. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard number 15437:2001, International Organization for Standardization — Information Technology, Genève, 2001.
17. M. E. Jiague, M. Frappier, F. Gervais, P. Konopacki, R. Laleau, J. Milhau, R. St-Denis. Model-Driven Engineering of Functional Security Policies. In *Proc. of ICEIS*, pages 374-379, 2010.
18. R. Mateescu, D. Thivolle. A model checking language for concurrent value-passing systems. In *Proc. of FM*, LNCS vol. 5014, pages 148–164, Springer, 2008.
19. J. Milhau, A. Idani, R. Laleau, M.A. Labiadh, Y. Ledru, M. Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. In *Journal of Innovations in Systems and Software Engineering*, vol. 7, pages 303-313, Springer, 2011.
20. D. Vekris, C. Dima. Efficient Operational Semantics for EB^3 for Verification of Temporal Properties. In *Proc. of FSEN*, Springer, 2013, to appear.